Summer 7-1-1986

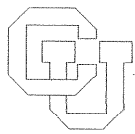
Demons, Catastrophes and Communicating Processes ; CU-CS-343-86

Michael G. Main
University of Colorado Boulder

Trace, Failure and Testing Equivalences for Communicating Processes *

Michael G. Main

CU-CS-343-86



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This research has been supported in part by National Science Foundation grant DCR-8402341.

TRACE, FAILURE AND TESTING EQUIVALENCES FOR COMMUNICATING PROCESSES*

Michael G. Main

CU-CS-343-86 July 1986
(Revised August 1987)

*This research has been supported in part by National Science Foundation grant DCR-8402341.

TRACE, FAILURE AND TESTING EQUIVALENCES FOR COMMUNICATING PROCESSES*

Michael G. Main
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA
Phone: 303-492-7579

ABSTRACT

A basic question in the theory of communicating processes is "When should two processes be considered equivalent?". Attempts to answer this question have led to the concepts of observation equivalence, bisimulations, testing equivalence, failure equivalence, etc. The main point of this paper is to increase the understanding and motivation for two of these equivalences, namely failure and testing equivalences. The approach starts with the idea that the equivalence of processes should be reducible to the visible sequences of actions which a process performs in various contexts. This idea is implemented by a string-based semantic order for communicating processes where divergence is catastrophic. Under some assumptions about contexts, the resulting semantics is shown to be equivalent to the *improved failure semantics* of Brookes and Roscoe [7] and also to the *must testing-semantics* of Hennessy and DeNicola [11,17,18]. This characterization gives independent support for the appropriateness of failures and testing.

KEYWORDS: Communicating processes, equivalences, failures, testing equivalence.

---

*This research has been supported in part by National Science Foundation grant DCR-8402341.

1. Introduction

Demonic nondeterminism is an order-theoretic approach to the semantics of nondeterministic processes. The approach embodies two principles:

The Partial Order in Demonic Nondeterminism:

There is a partial order on processes, where a relationship $p \sqsubseteq q$ means that less can be guaranteed about the behavior of p than about the behavior of q. Less nondeterminism in a process results in a higher position in this partial order. In terms of program correctness: if we guarantee that every possible behavior of a process is correct, then this process can always be replaced by a process which is higher in the semantic order, and the result will still be correct. The least element in this order is a process called *chaos*, whose behavior is a nondeterministic choice between all possible behaviors.

Catastrophe Principle:

In some settings, there are behaviors which are always bad. For example, most programmers would agree that an infinite loop with no input/output is always undesirable. Or, in Milner's Calculus of Communicating Systems, we might wish to avoid infinite sequences of internal actions. The term *divergence* refers to such a behavior, which is always to be avoided. According to the *catastrophe principle*, a process which *might* diverge is just as bad as a process which always diverges. Hence, all processes which have the nondeterministic possibility of divergence are identified with each other. In fact, they are all identified with *chaos*, since one of the many nondeterministic choices of *chaos* is divergence. In terms of program correctness, this principle assumes that divergence is a behavior that we must always avoid. Hence a process which might diverge is always incorrect, and it does no further harm to identify it with the catastrophic behavior of *chaos*.

This approach to nondeterminism is appropriate if we are studying properties which hold for all behaviors of a process. However, it does not capture all information about a process. For example, the approach cannot indicate which behaviors cannot occur in a process that has the possibility of diverging, since such a process has been identified with chaos. This also occurs with Dijkstra's weakest precondition semantics [10] and the Smyth powerdomain [21], both of which incorporate

these two principles. More recently, the two principles have explicitly justified various approaches to nondeterminism (for example, [2,4,5,9,11,17,18,19,20]). Of particular note are two applications of the principles to the semantics of communicating processes:

In *failure semantics*, proposed by Brookes, Hoare and Roscoe [6,7], a process is characterized by a set of pairs called *failures*. A failure (s,X) consists of a string of actions s which a process can perform, and a set X of actions which the process can refuse to perform immediately after doing s. A higher position in the semantic order corresponds to fewer failures.

The *must* semantics is one of three testing semantics proposed by Hennessy and DeNicola [11,17,18]. A process is characterized by defining a set of tests which it might not pass. A higher position in the semantic order corresponds to having fewer tests which might not be passed.

Both these approaches have provided models of communicating processes. But, a basic question remains: *Why are failures important?* — or — *Why are these tests used?* These questions can only be answered by appealing to some concept which is widely accepted as fundamental on its own merit. The concept chosen here is a string of actions. Of course, the purity of this concept is subjective, but it can be justified as underlying most operational models of processes, as well as automata theory and formal language theory. In any case, we can define a semantic order where a higher position corresponds to being able to do fewer strings of actions.

In general, the string-based order is less discriminating than the other two orders. However, none of these orders are used in their raw form. Instead, they are closed under contexts, so that in the final form, a process is lower than another only if this lower-than relationship remains valid for all contexts in which the processes may be placed. (If this closure is not explicitly done, it is because the order is already closed, which is the case for the failures order on CSP processes.)

The new result here is this: under some assumptions about contexts, all three closed orders are identical (failures, must-testing and strings).

A familiarity with Milner's Calculus of Communicating Systems (CCS) would be useful (but not vital) in reading the paper. Section 2 defines the various semantic orders on processes, based on failures, must-testing, and strings. This is done using an abstract model of processes called *labeled transition systems*, which underlies CCS and other models of communicating processes. Section 3 provides the basic results about when the orders are identical. Section 4 gives examples of labeled transition systems for which the results apply.

2. Order-Semantics of Labeled Transition Systems

2.1 Labeled Transition Systems

A labeled transition system consists of two kinds of objects – *processes* and *actions* – and relations which indicate how actions can transform one process into another. These systems are the abstract models which underlie many studies of communicating processes, such as Keller's original study of parallel programs [14], Milner's *Calculus of Communicating Systems* [15] and Hoare's *Communicating Sequential Processes* [13]. The formal definition given below is closest to DeNicola's systems [17].

Definition 2.1. A *labeled transition system* (LTS) is a set P of *processes* and an infinite set A of *actions*, together with these:

- *The internal (or invisible action):* A special element of A, denoted by τ.
- *Transition Relations:* For each action $\alpha \in A$ there is a binary relation $\xrightarrow{\alpha}$ on P. (If p is related to q by $\xrightarrow{\alpha}$ then we write $p \xrightarrow{\alpha} q$.)
- *Divergence Predicate:* A predicate $\uparrow$ on the set P of processes. (When the predicate is *true* for a process p then we write $p\uparrow$; otherwise we write $p\downarrow$.) The predicate meets the property that for all $p, q \in P$: whenever $(p \xrightarrow{\tau} q)$ and $q\uparrow$, then also $p\uparrow$. □

Intuitively, $p\uparrow$ means that the process p is capable of diverging, in the sense defined in Section 1. The relationship $p \xrightarrow{\alpha} q$ means that the action α is capable of transforming the process p to the process q. The internal action τ is not observable by an outside observer.

Visible (non-τ) actions are called *events*. In general we use p, q and r for arbitrary processes and Q for a set of processes. Lower case Greek letters are arbitrary actions (with τ the internal action); s and t range over finite strings of events; X ranges over finite subsets of events.

2.2 Notation about Transitions

The notation $p_0 \xrightarrow{\alpha_1} p_1 \xrightarrow{\alpha_2} p_2 \cdots p_{n-1} \xrightarrow{\alpha_n} p_n$ is used as shorthand for $p_0 \xrightarrow{\alpha_1} p_1$ and $p_1 \xrightarrow{\alpha_2} p_2$ and $\cdots p_{n-1} \xrightarrow{\alpha_n} p_n$. We also use the following notation:

$p \Longrightarrow q$

means there is a sequence of processes $p = p_0 \xrightarrow{\tau} p_1 \xrightarrow{\tau} p_2 \cdots p_{n-1} \xrightarrow{\tau} p_n = q$.

$p \overset{\alpha}{\Longrightarrow} q$

means there are processes such that $p \Longrightarrow p_0 \xrightarrow{\alpha} p_1 \Longrightarrow q$.

$p \overset{s}{\Longrightarrow} q$

means there is a sequence of processes $p_0 \overset{\alpha_1}{\Longrightarrow} p_1 \overset{\alpha_2}{\Longrightarrow} p_2 \cdots p_{n-1} \overset{\alpha_n}{\Longrightarrow} p_n$, where $p = p_0$, $q = p_n$ and $s = \alpha_1 \alpha_2 \cdots \alpha_n$.

$p \overset{s}{\Longrightarrow}$

means there is a process q such that $p \overset{s}{\Longrightarrow} q$.

Stopped processes

A process p is *stopped* if $p\downarrow$ and there is no event α and process q such that $p \xrightarrow{\alpha} q$. (Hence, a stopped process cannot diverge, nor can it be transformed by any events.)

$p\uparrow s$, and $p\downarrow s$

The first expression means there is some process q and some string t such that t is a prefix of s and $p \overset{t}{\Longrightarrow} q$ and $q\uparrow$. The second means there are no such q and t. The intuitive meaning of $(p\uparrow s)$ is that divergence may occur if we try to transform the process p using the string s.

(p **after** s)

This is the set of processes $\{q \mid p \stackrel{s}{\Longrightarrow} q\}$.

(p **must** X) and (Q **must** X)

The first expression means that whenever $p \Longrightarrow q$, then there exists some $\alpha \in X$ such that $q \stackrel{\alpha}{\Longrightarrow}$. The second expression (Q **must** X) means that (p **must** X) holds for each $p \in Q$.

Finite Choice LTS

An LTS is called *finite choice* provided that for every process p and string s:

$$p \downarrow s \text{ implies that the set } \{\alpha \in A \mid p \stackrel{s\alpha}{\Longrightarrow}\} \text{ is finite.}$$

(This is a weaker version of Abramsky's *sort-finiteness* [1].)

2.3 Order-Theoretic Semantics

We can put several semantic orders on the processes P of an arbitrary LTS. These orders are motivated by the principles of Section 1. Intuitively, an order-relation $p \sqsubseteq q$ means that p is "more nondeterministic" than q. This will be obvious in the first order presented below, but perhaps less obvious in the others. The orders are actually *pre-orders*, meaning that they are transitive, reflexive relations – but not necessarily anti-symmetric. Thus, we may have $p \sqsubseteq q$ and also $q \sqsubseteq p$, which means that p and q are equally nondeterministic.

Definition 2.2. TRACE pre-order:

The *traces* of a process are the finite strings of events which a process may undergo, ending in a stopped process. The process called *chaos* is one which may undergo every possible trace. In the setting of the Catastrophe Principle, a process which diverges is identified with *chaos*. This suggests that there are two kinds of traces of a process p: (1) If $p \stackrel{s}{\Longrightarrow} q$ and q is stopped, then s is a trace of p; (2) If $p \stackrel{s}{\Longrightarrow} q$ and $q\uparrow$, then st is a trace of p (for every t). Formally the set $TRACE(p)$ is:

$$\{s \mid p \overset{\wedge}{\downarrow} s, \text{ or for some } q: p \overset{s}{\Longrightarrow} q \text{ and } q \text{ is stopped}\}.$$

The trace pre-order on processes is defined by:

$$p \sqsubseteq^{\text{TRACE}} q \text{ iff } TRACE(p) \supseteq TRACE(q).$$

Definition 2.3. EMPTY STRING pre-order:

This pre-order is less discriminating than the trace pre-order. Its motivation will become evident in Section 3. We denote the empty string by ε.

$$p \sqsubseteq^{\text{ES}} q \text{ iff } (\varepsilon \text{ is in } TRACE(q) \text{ implies } \varepsilon \text{ is in } TRACE(p)).$$

Definition 2.4. MUST pre-order:

This was defined by DeNicola and Hennessy [11,17,18] as one of three pre-orders based on testing. For our purposes, an alternate characterization given by DeNicola and Hennessy is easier to work with. Here is the definition (from [18, Definition 6.4.1]):

$p \sqsubseteq^{\text{MUST}} q$ iff for all s, X: $p \overset{\vee}{\downarrow} s$ implies

(i) $q \overset{\vee}{\downarrow} s$, and

(ii) ((p **after** s) **must** X) implies ((q **after** s) **must** X).

Definition 2.5. FAILURE pre-order:

This was first defined by Brookes, Hoare and Roscoe for CSP processes [5,6,7], and later by Brookes for CCS [4]. The definition depends on *failures*, which are pairs (s,X), where s is a string of events, and X is a finite set of events. The definition given here corresponds to the failure semantics with divergence, used by [7]. Intuitively, $(s,X) \in FAILURE(p)$ means that after p undergoes the string s, it might diverge or be unable to perform an event from X. Here is the formal definition of failures, and the failure pre-order:

$$FAILURE(p) = \{(s,X) \mid p \overset{\wedge}{\downarrow} s, \text{ or } \textit{not } ((p \textbf{ after } s) \textbf{ must } X)\}.$$

$$p \sqsubseteq^{\text{FAILURE}} q \text{ iff } FAILURE(p) \supseteq FAILURE(q).$$

2.3 CONTEXT pre-orders:

In general, it is not sufficient to know that a process is more nondeterministic than another. Instead, we need to show that this relationship remains valid for whatever context the processes find themselves in. To formalize this, we define an LTS with (unary) contexts:

Definition 2.6. *An LTS with contexts* is an LTS together with a set of contexts. Each context is a function $C : P \rightarrow P$ on the set P of processes, and the set of contexts is closed under composition. Thus, if $C_1[\ ]$ and $C_2[\ ]$ are contexts, then so is their composition $C_2[C_1[\ ]]$. □

If we show that for every context C, $C[p]$ is more nondeterministic than $C[q]$, then whenever $C[p]$ is correct, so is $C[q]$. Intuitively: it is always possible to replace an instance of p by q — without destroying correctness of the surrounding program. This is the basis of four new pre-orders:

$(p \sqsubseteq_C^{\text{TRACE}} q)$ iff for all contexts C: $(C[p] \sqsubseteq^{\text{TRACE}} C[q])$.

$(p \sqsubseteq_C^{\text{ES}} q)$ iff for all contexts C: $(C[p] \sqsubseteq^{\text{ES}} C[q])$.

$(p \sqsubseteq_C^{\text{FAILURE}} q)$ iff for all contexts C: $(C[p] \sqsubseteq^{\text{FAILURE}} C[q])$.

$(p \sqsubseteq_C^{\text{MUST}} q)$ iff for all contexts C: $(C[p] \sqsubseteq^{\text{MUST}} C[q])$.

3. Results on the Pre-orders

The eight pre-orders of Section 2 are the topic of this section. The results apply mostly to finite-choice LTS's. In these systems we show:

$$(p \sqsubseteq^{\text{MUST}} q) \text{ iff } (p \sqsubseteq^{\text{FAILURE}} q) \text{ implies } (p \sqsubseteq^{\text{TRACE}} q) \text{ implies } (p \sqsubseteq^{\text{ES}} q).$$

These results are used in Section 4 to show that the four context pre-orders coincide for commonly used LTS's with contexts, such as CCS and CSP. We begin with some preliminary lemmas.

Lemma 3.1. *Let s be a string of events and p be a process in a finite-choice LTS. Then $s \in TRACE\ (p)$ iff for all finite sets X of events, $(s, X) \in FAILURE\ (p)$.*

Proof: (A) Assume $s \in TRACE\ (p)$. We must show that for every finite set X of events: $(s, X) \in FAILURE\ (p)$. If $p \uparrow s$ then $(s, X) \in FAILURE\ (p)$ is immediate. On the other hand, if $p \downarrow s$ then there is some process q such that $p \stackrel{s}{\Longrightarrow} q$ and q is stopped. Let q be this process and note that (since q is stopped), for every event α: *not* $(q \stackrel{\alpha}{\Longrightarrow})$. Therefore for every finite set X of events: *not* $(q$ **must** $X)$. This implies $(s, X) \in FAILURE\ (p)$, as required.

(B) Assume that for all finite sets X of events: $(s, X) \in FAILURE\ (p)$. We must show that $s \in TRACE\ (p)$. If $p \uparrow s$ then $s \in TRACE\ (p)$ is immediate. Hence we assume $p \downarrow s$. Consider the set $X = \{\alpha \in A \mid p \stackrel{s\alpha}{\Longrightarrow}\}$. Since the LTS is finite-choice, this set is finite, so $(s, X) \in FAILURE\ (p)$. This implies that there is at least one process q such that $p \stackrel{s}{\Longrightarrow} q$ and *not* $(q$ **must** $X)$. Let q be such a process which minimizes the size of the finite set $Y = \{\alpha \in A \mid q \stackrel{\alpha}{\Longrightarrow}\}$ (a subset of X). Notice that the minimality condition implies that whenever $q \Longrightarrow r$, then $\{\alpha \in A \mid r \stackrel{\alpha}{\Longrightarrow}\} = Y$. Hence, if Y is nonempty, then $(q$ **must** $Y)$, which implies $(q$ **must** $X)$. But since *not* $(q$ **must** $X)$, it follows that Y is empty; in other words, q is stopped, and so $s \in TRACE\ (p)$. □

Lemma 3.2. *Let s be a string of events and p be a process in a finite-choice LTS. Then*

$$p\uparrow s \quad \textit{iff} \quad \text{for all } t\text{: } st \in TRACE\ (p).$$

Proof: (A) Suppose $p\uparrow s$, and let t be a string of events. Then $p\uparrow st$, which implies $st \in TRACE\ (p)$.

(B) Suppose for every string of events t, that $st \in TRACE\ (p)$. Hence $\{\alpha |\ p \stackrel{s\alpha}{\Longrightarrow}\}$ includes every event, hence it is infinite. Since the LTS is finite-choice, this implies $p\uparrow s$. □

The fact that there are an infinite number of events is critical to the proof of part (B) in Lemma 3.2. This is the only place in the paper where this fact is used. If we are willing to accept Lemma 3.2 as an axiom, then the results of this section apply to any LTS, whether or not the event set is infinite.

Lemma 3.3. *Let p and q be processes in a finite-choice LTS. Then $(p \sqsubseteq^{\text{FAILURE}} q)$ implies $(p \sqsubseteq^{\text{TRACE}} q)$.*

Proof: Assume $p \sqsubseteq^{\text{FAILURE}} q$. Then for every string s of events:

$s \in TRACE\ (q)$ implies for all X: $(s,X) \in FAILURE\ (q)$ (by Lemma 3.1)

implies for all X, $(s,X) \in FAILURE\ (p)$ (since $p \sqsubseteq^{\text{FAILURE}} q$)

implies $s \in TRACE\ (p)$ (by Lemma 3.1)

Therefore, $TRACE\ (p) \supseteq TRACE\ (q)$ and $p \sqsubseteq^{\text{TRACE}} q$. □

Lemma 3.4. *Let p and q be processes in a finite-choice LTS. Then $(p \sqsubseteq^{\text{FAILURE}} q)$ if and only if $(p \sqsubseteq^{\text{MUST}} q)$.*

Proof: (A) Assume $(p \sqsubseteq^{\text{FAILURE}} q)$. To prove $(p \sqsubseteq^{\text{MUST}} q)$, let s be a finite string of events with $p \downarrow s$, and let X be a finite subset of events. We must prove that (i) $(q \downarrow s)$, and also (ii) $((p \textbf{ after } s) \textbf{ must } X)$ implies $((q \textbf{ after } s) \textbf{ must } X)$:

(i) From Lemma 3.2, there exists a string t such that $st \notin TRACE\ (p)$. From Lemma 3.3, this implies that st is also not in $TRACE\ (q)$. Hence (again by Lemma 3.2), we have $q \downarrow s$.

(ii) Assume $((p \textbf{ after } s) \textbf{ must } X)$ so that $(s, X) \notin FAILURE\ (p)$. Since $p \sqsubseteq^{\text{FAILURE}} q$, this implies that (s, X) is also not in $FAILURE\ (q)$. This (together with $q \downarrow s$) implies that $((q \textbf{ after } s) \textbf{ must } X)$, as required.

(B) Assume $(p \sqsubseteq^{\text{MUST}} q)$ and suppose that $(s, X) \notin FAILURE\ (p)$. To prove $(p \sqsubseteq^{\text{FAILURE}} q)$ we must show that $(s, X) \notin FAILURE\ (q)$. Now, if $(p \uparrow s)$ then $(s, X) \in FAILURE\ (p)$ is immediate (by Definition 2.5). Hence, we need only consider the case when $(p \downarrow s)$, which also implies $(q \downarrow s)$. In this case we have $((p \textbf{ after } s) \textbf{ must } X)$, which implies $((q \textbf{ after } s) \textbf{ must } X)$, and this implies that $(s, X) \notin FAILURE\ (q)$, as required. □

Lemma 3.5. *Let p and q be processes in an LTS. Then $(p \sqsubseteq^{\text{TRACE}} q)$ implies $(p \sqsubseteq^{\text{ES}} q)$.*

Proof: Immediate from the definitions of the trace and empty-string pre-orders. □

Theorem 3.6. *Let L be a finite-choice LTS system with contexts, such that the following property holds:*

For all s and X, there exists a context C_{sX}, such that for all processes p,

$$[(s, X) \in FAILURE\ (p)] \quad \textit{iff} \quad [\varepsilon \in TRACE\ (C_{sX}[p])].$$

Then the pre-orders $(\sqsubseteq_c^{\text{TRACE}}, \sqsubseteq_c^{\text{ES}}, \sqsubseteq_c^{\text{FAILURE}}, \sqsubseteq_c^{\text{MUST}})$ are identical in L.

Proof: From the previous lemmas, we have the following results (where ≤ means the relation on the left is smaller):

$\sqsubseteq^{\text{MUST}}$ equals $\sqsubseteq^{\text{FAILURE}}$ (Lemma 3.4) hence also $\sqsubseteq_c^{\text{MUST}}$ equals $\sqsubseteq_c^{\text{FAILURE}}$

$$\sqsubseteq^{\text{FAILURE}} \leq \sqsubseteq^{\text{TRACE}} \text{ (Lemma 3.3) hence also } \sqsubseteq_c^{\text{FAILURE}} \leq \sqsubseteq_c^{\text{TRACE}}$$

$$\sqsubseteq^{\text{TRACE}} \leq \sqsubseteq^{\text{ES}} \text{ (Lemma 3.5) hence also } \sqsubseteq_c^{\text{TRACE}} \leq \sqsubseteq_c^{\text{ES}}$$

It only remains to show that ($\sqsubseteq_c^{\text{ES}} \leq \sqsubseteq_c^{\text{FAILURE}}$), so assume $p \sqsubseteq_c^{\text{ES}} q$ for some processes p and q. Then for every context D, string s, and finite set X of events, we have:

$$(s, X) \in \mathit{FAILURE}\ (D[q])$$

iff

$$\varepsilon \in \mathit{TRACE}\ (C_{sX}[D[q]])$$

implies

$$\varepsilon \in \mathit{TRACE}\ (C_{sX}[D[p]])$$

iff

$$(s, X) \in \mathit{FAILURE}\ (D[p]).$$

This is just $p \sqsubseteq_c^{\text{FAILURE}} q$. Therefore $\sqsubseteq_c^{\text{ES}} \leq \sqsubseteq_c^{\text{FAILURE}}$. □

4. CCS and CSP Processes

This section shows that Milner's CCS [15] is an LTS with contexts that meets the requirements of Theorem 3.6. Therefore, the four context pre-orders are identical on these processes. The same result is shown to hold for the operational model of CSP presented by Brookes, Roscoe and Walker [8].

4.1. The Finite-Choice LTS Formed by CCS Processes

The CCS model used here is taken from DeNicola and Hennessy's paper [18] (where they introduced the MUST pre-order). To make this paper more self-contained, these definitions are recapitulated here, with excerpts from [18]. If you are already familiar with these definitions, then skip to Lemma 4.1.

Actions: Let Δ be an infinite set, and for each $\alpha \in \Delta$, let $\bar{\alpha}$ be a new element not in Δ. We define $\bar{\Delta}$ to be $\{\bar{\alpha} \mid \alpha \in \Delta\}$. We extend the $\bar{}$ operation to all of $\Delta \cup \bar{\Delta}$ by defining $\bar{\bar{\alpha}}$ to be α. The set of actions is the set $\Delta \cup \bar{\Delta} \cup \{\tau\}$. We assume that τ is not in Δ or $\bar{\Delta}$. A *relabeling* is a partial function $R : A \rightarrow A$ such that $R(\tau) = \tau$, and for every event α: if $R(\alpha)$ is defined then so is $R(\bar{\alpha})$, and $\overline{R(\alpha)} = R(\bar{\alpha})$.

Processes: Process are closed CCS terms, which are defined using the set A of actions and an infinite set V of variables. Before we define processes, we give a recursive definition of *process expressions*:

A. *NIL* and Ω are process expressions. (Intuitively, *NIL* is a process which cannot perform an action; Ω is a process that always diverges.)

B. If p is a process expression and α is an action, then (αp) is a process expression. (Intuitively, αp is a process which can transform into p via an α action.)

C. If p is a process expression and S is a relabeling, then $S(p)$ is a process expression. (Intuitively, $S(p)$ is like p, but its actions have been relabeled according to S. DeNicola and Hennessy use the notation $p[S]$, which we avoid to eliminate confusion with the notation for contexts.)

D. If p and q are process expressions, then so are $(p+q)$ and $(p \mid q)$. (Intuitively, $p+q$ is an external nondeterministic choice between p and q. It can perform actions of either p or q. The process $p \mid q$ results from running p and q in parallel, and allowing them to communicate.)

E. Each variable x is a process expression. If p is a process expression and x is a variable, then **rec** $x.p$ is also a process expression. (Intuitively, this is a recursive process, defined as the solution to the equation $x = p$.) □

The operation **rec** $x.t$ binds occurrences of x in the subterm t. This gives rise to the usual notions of free and bound variables in a term. The processes which we will use in this section are the process expressions with no free variables, called *closed process expressions*. As usual, operations in process expressions are given precedences. Concatenation of an action on the left (as in αp) has highest precedence, followed by $|$, + and **rec** x (in that order). These precedences will be used to omit parenthesis when possible. Also, occurrences of *NIL* are usually omitted so that $((\alpha NIL)+(\beta NIL))$ is written $\alpha+\beta$.

Transitions: For any action α, the transition relation $\xrightarrow{\alpha}$ is the relation from Definition 2.1.1 of DeNicola and Hennessy's paper, and repeated here: Let $\xrightarrow{\alpha}$ be the least relation over closed terms which satisfies:

(i) $\alpha p \xrightarrow{\alpha} p$;

(ii) $p_1 \xrightarrow{\alpha} q$ implies
$$\begin{cases} p_1+p_2 \xrightarrow{\alpha} q, \\ p_2+p_1 \xrightarrow{\alpha} q, \\ p_1 | p_2 \xrightarrow{\alpha} q | p_2 \\ p_2 | p_1 \xrightarrow{\alpha} p_2 | q; \end{cases}$$

(iii) $p \xrightarrow{\alpha} q$ and $S(\alpha)$ defined, implies $S(p) \xrightarrow{S(\alpha)} S(q)$;

(iv) $p_1 \xrightarrow{\alpha} q_1$ and $p_2 \xrightarrow{\bar{\alpha}} q_2$, implies $p_1 | p_2 \xrightarrow{\tau} q_1 | q_2$;

(v) Let t' be the expression obtained from t by replacing every free occurrence of x by **rec** $x.t$.

If $t' \xrightarrow{\alpha} q$, then $\mathbf{rec}\, x.t \xrightarrow{\alpha} q$.

Divergence Predicate: The divergence predicate is the predicate $\Uparrow$ defined at the bottom of page 91 in DeNicola and Hennessy's paper. It is straightforward to show that it meets the requirement in Definition 2.1 for a divergence predicate. The results of this section depend only on this requirement and on the following observation about the divergence predicate:

$$\text{if } p_0 \xrightarrow{\tau} p_1 \xrightarrow{\tau} p_2 \xrightarrow{\tau} \cdots \text{ is an infinite CCS computation sequence, then } p_0 \Uparrow.$$

This property will be used in the proof of Lemma 4.1.

Contexts: Contexts are CCS process expressions, formed with one free variable (see page 92 [18] or Definition 2.2.3 in [17]). For example, the CCS term $\beta + \alpha x$ is the context with $C[p] = \beta + \alpha p$, for any process p. These contexts are closed under composition.

With the above definitions, CCS processes form an LTS with contexts. The next lemma demonstrates the fact that this LTS is finite-choice.

Lemma 4.1. *(CCS is a finite-choice LTS.) Let p be a CCS process and s a string of events, such that* $B = \{\alpha \mid p \stackrel{s\alpha}{\Longrightarrow}\}$ *is infinite. Then* $p \Uparrow s$.

Proof: Note that the set B is equal to $\{\alpha \mid \exists r : p \stackrel{s}{\Longrightarrow} r \text{ and } r \xrightarrow{\alpha} \text{ and } \alpha \neq \tau\}$. Also, an induction on the definition of processes shows that for any process r, the set $\{\alpha \mid r \xrightarrow{\alpha}\}$ is finite. Therefore, since B is infinite, the set $\{r \mid p \stackrel{s}{\Longrightarrow} r\}$ is also infinite. Thus, there must be arbitrarily long CCS computation sequences of the form

$$p \xrightarrow{\alpha_1} p_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} p_n$$

where the visible part of $\alpha_1 \alpha_2 \cdots \alpha_n$ is s. (These must become arbitrarily long because if their length were bounded then there would be only a finite number of different sequences and the set $\{r \mid p \stackrel{s}{\Longrightarrow} r\}$ would be finite.) If these sequences become arbitrarily long, then they must also con-

tain arbitrarily long subsequences which contain only τ-moves. This implies that there is a process q and a prefix t of s such that there are arbitrarily long CCS computation sequences of the form

$$p \stackrel{t}{\Longrightarrow} q \stackrel{\tau}{\longrightarrow} q_1 \stackrel{\tau}{\longrightarrow} q_2 \cdots .$$

This implies that q is the starting point of an infinite CCS computation sequence of the form $q \stackrel{\tau}{\longrightarrow} q_1 \stackrel{\tau}{\longrightarrow} q_2 \stackrel{\tau}{\longrightarrow} \cdots$. As pointed out above, this implies $q\uparrow$. Hence, $p\uparrow t$ which implies $p\uparrow s$.

□

4.2 The Four Context Pre-orders on CCS Processes

In order to show that the four context pre-orders are identical on CCS processes, we only need to demonstrate the hypothesis of Theorem 3.6. For this purpose, let s be any string of events and let X be a finite set of events. We must find a context C_{sX} such that for any process p: $(s, X) \in FAILURE\ (p)$ if and only if $\varepsilon \in TRACE\ (C_{sX}[p])$. This context is defined as follows:

- Let α be some event such that neither α nor $\bar{\alpha}$ appear in the set X or the string s.
- Let $S : A \rightarrow A$ be any relabeling function which is totally defined, is the identity on events of X and s, and such that neither α nor $\bar{\alpha}$ is in the image of S.
- Let $T : A \rightarrow A$ be the partial function which is the identity for τ, α and $\bar{\alpha}$, and undefined everywhere else.
- Let $\alpha_1 \cdots \alpha_n = s$ be the individual events of s, let X be the set $\{\chi_1, \cdots, \chi_k\}$, and define two processes r_1 and r_2 by the synchronization trees in Figures 1 and 2. (Note that $r_1 \stackrel{\bar{s}}{\Longrightarrow} r_2$.)

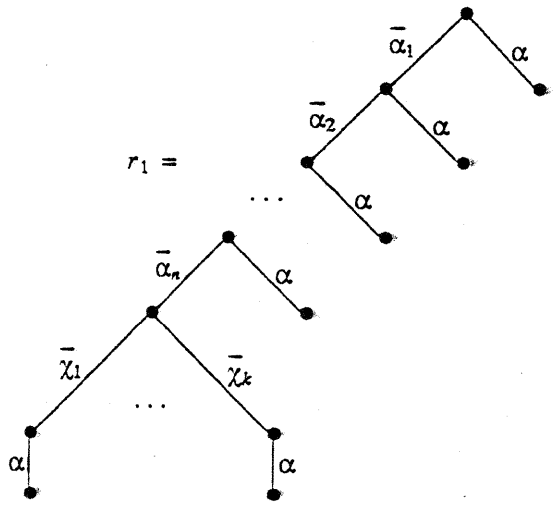


FIGURE 1.

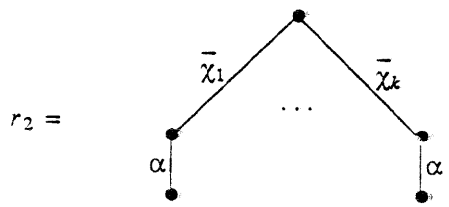


FIGURE 2.

- Let $C_{sX}:P \rightarrow P$ be the context defined by $C_{sX}[p] = T(S(p) \mid r_1)$.

The next two lemmas show that the context C_{sX} meets the iff requirement stated above.

Lemma 4.2. *Let p be any CCS process. If $\varepsilon \in TRACE\ (C_{sX}[p])$ then $(s,X) \in FAILURE\ (p)$.*

Proof: If $p\uparrow s$, then $(s,X) \in FAILURE\ (p)$ is immediate from Definition 2.5. So, we assume $p\downarrow s$, which also implies $C_{sX}[p]\downarrow$. From this (and Definition 2.2), it follows that the only way for $\varepsilon \in TRACE\ (C_{sX}[p])$ to hold is if there is a stopped process q such that $C_{sX}[p] \Longrightarrow q$. From the definition of CCS transitions (Definition 2.1.1 in DeNicola and Hennessy), q must have the form $q = T(S[p'] \mid r')$ for some processes p' and r', where there is some string t with $S[p] \stackrel{t}{\Longrightarrow} S[p']$ and $r_1 \stackrel{\bar{t}}{\Longrightarrow} r'$. (Here $\bar{t}$ is obtained from t by replacing each event in t by its complement, as usually defined in CCS [18, page 89].) Note these facts about p', r', and t:

Fact 1: *not* $(r' \stackrel{\alpha}{\Longrightarrow})$ — since otherwise q is not stopped.

Fact 2: The transformation $r_1 \stackrel{\bar{t}}{\Longrightarrow} r'$ does not include any α event — since otherwise t must contain an $\bar{\alpha}$ event which cannot occur in a transformation $S[p] \stackrel{t}{\Longrightarrow} S[p']$. (Recall that $\bar{\alpha}$ is not in the image of S.)

Fact 3: $r' = r_2$ — since $r' = r_2$ is the only process such that $r_1 \stackrel{\bar{t}}{\Longrightarrow} r'$, which meets both Facts 1 and 2.

Fact 4: For all $\beta \in X$: *not* $(p' \stackrel{\beta}{\Longrightarrow})$ — otherwise q is not stopped. (Since $r' = r_2$, q would be able to do a communication step between p' and r', followed by an α event).

Now, $\bar{t} = \bar{s}$ is immediate from Fact 3, and this implies that $t = s$. Hence $S[p] \stackrel{s}{\Longrightarrow} S[p']$. Since S is the identity on events of s, this implies $p \stackrel{s}{\Longrightarrow} p'$. Finally, this (together with Fact 4) implies that $(s,X) \in FAILURE\ (p)$, as required. □

Lemma 4.3. *Let p be any process. If $(s, X) \in FAILURE\ (p)$ then $\varepsilon \in TRACE\ (C_{sX}[p])$.*

Proof:

Case 1: $p \downarrow s$. In this case, there is some process p' such that $p \stackrel{s}{\Longrightarrow} p'$ and *not* (p' **must** X). From the definition of **must**, there is a process q such that $p' \Longrightarrow q$ and *not* ($q \stackrel{\alpha}{\Longrightarrow}$) for any $\alpha \in X$. Now, q cannot be transformed by any event of X, so neither can $S(q)$ since S is the identity on X. This implies that $T(S(q) \mid r_2)$ is stopped. Finally, since $r_1 \stackrel{\bar{s}}{\Longrightarrow} r_2$, we have:

$$C_{sX}[p] = T(S(p) \mid r_1) \Longrightarrow T(S(p') \mid r_2) \Longrightarrow T(S(q) \mid r_2).$$

The last process in this derivation is stopped, therefore $\varepsilon \in TRACE\ (C_{sX}[p])$, as required.

Case 2: $p \uparrow s$. In this case there is some decomposition of s, say $s = s_1 s_2$, such that $p \stackrel{s_1}{\Longrightarrow} p'$ and $(p') \uparrow$ (for some process p'). From the definition of CCS transitions, it follows that

$$C_{sX}[p] = T(S(p) \mid r_1) \Longrightarrow T(S(p') \mid r'),$$

where r' is the process such that $r_1 \stackrel{\bar{s}_1}{\Longrightarrow} r'$. Moreover, $(p') \uparrow$ implies $T(S(p') \mid r') \uparrow$, hence also $C_{sX}[p] \uparrow$. By Definition 2.2, this implies that $\varepsilon \in TRACE\ (C_{sX}[p])$, as required. □

From these two lemmas and Theorem 3.6, follows the main result of this section:

Theorem 4.4. *The four context pre-orders are identical in CCS processes.*

4.3. CSP Processes

A model of Hoare's communicating sequential processes (CSP) based on failures was proposed by Brookes, Hoare and Roscoe [6]. Later this model was improved by Brookes and Roscoe to take divergence into account [7], and this model has been expressed as a labeled transition system by Brookes, Roscoe and Walker [8]. This labeled transition system meets the requirements of Theorem 3.6, so that that four context pre-orders are again identical. We provide the proof in more informal terms than the previous section on CCS.

The labeled transition system is formed by taking the set of closed CSP-terms as processes [8, Section 1]. The set of actions is the alphabet A^+ of visible actions plus τ which underlies the CSP-terms. As in Section 4.2 of this paper, we only consider the case where A^+ is infinite. For any action $\alpha \in A^+$, the transition relation $\xrightarrow{\alpha}$ is the relation which defined in Section 2 of [8] (and written with the same notation as here). The divergence predicate is defined by $p_0 \uparrow$ if and only if there exist processes $p_1, p_2, p_3 \cdots$ such that $p_0 \xrightarrow{\tau} p_1 \xrightarrow{\tau} p_2 \xrightarrow{\tau} \cdots$ is an infinite CSP computation sequence [8, Section 2]. Because of this property, the proof of Lemma 4.1 also applies to CSP processes: *i.e.*, CSP forms a finite-choice LTS. (The finite-choice property also follows from Lemma 4 in [8].) A context is a CSP-term formed with one free variable.

In order to show that the four context pre-orders are identical on CSP processes, we need to demonstrate the hypothesis of Theorem 3.6. For this purpose, let s be any string of events and let X be a finite set of events. We must find a context C_{sX} such that for any process p: $(s, X) \in FAILURE(p)$ if and only if $\varepsilon \in TRACE(C_{sX}[p])$. This context is similar to that given in Section 4.2, making use of some fixed event α which appears in neither s nor X; specifically, C_{sX} is defined by:

$$C_{sX}[p] = (S[p]_B \|_A r_1),$$

where A is the set of all events, and S, B, and r_1 are defined as follows: $S: A \to A$ is an alphabet transformation [8, Section 1] which is the identity on events of X and s, and such that α is not in the image of S. $B = A - \{\alpha\}$ is the set of all events except α. The process r_1 is a process with the synchronization tree of Figure 3 (where $\alpha_1 \cdots \alpha_n$ is the string s, and X is the set $\{\chi_1, \cdots, \chi_k\}$).

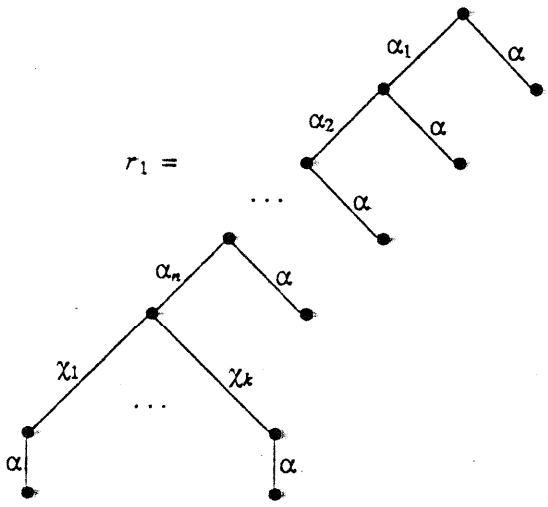


FIGURE 3.

The CSP synchronous parallel composition operation ${}_B\|_A$ has an effect similar to the restriction function T combined with the CCS parallel composition in Section 4.2. The proof that C_{sX} meets the hypothesis of Theorem 3.6 follows the same line taken in Section 4.2. Thus, the four context pre-orders are identical for this variant of CSP.

5. Discussion

The main result shows that for certain labeled transition systems (such as CCS and CSP), *failure* semantics and *must* testing semantics are identical to the semantics obtained by considering only traces of events that a process can perform in various contexts. In fact, this equivalence holds if we only consider the empty trace. Several similar results have appeared recently. Pnueli [19] states the equivalence of trace semantics and the *must* testing semantics for a CCS-like language without recursion, restriction, relabeling, or the silent τ-action. A recent paper by DeBakker, Meyer and Olderog [3] shows an equivalence between a stream-based order (similar to traces) and an order based on *observations*. In both of these cases, an appeal is made to the naturalness of trace semantics – and Hennessy makes a similar statement in the title of a recent note: "Why Testing Equivalence is Natural" [12]. In this note Hennessy shows that a variety of pre-orders (including one based on traces) coincide with the MUST pre-order for certain CCS processes. To quote Pnueli [19] on traces:

> "An argument that we may advance in order to claim that trace congruence is the right equivalence relation is that when we observe a complete system from outside, the only observable behavior is given by the maximal traces. Specifying an internal module, we only need as much detail that will guarantee replaceability. That is, two alternate modules that agree on this level of detail should be completely interchangeable in the sense that exchanging one by the other will not change the external behavior. Hence trace congruence seems to be precisely the relation we are looking for."

Thus, the main result of this paper is both a characterization and a justification for *failures* and *must* testing.

The conditions under which the main result holds are explicitly stated in Section 3. It seems reasonable to expect other variations on CCS to meet these conditions. In particular, the various fair notions of parallel merge may meet these conditions, although care must be taken that the resulting LTS is finite-choice. The synchronous merge of SCCS [16] also may be modified to meet these conditions.

ACKNOWLEDGEMENTS

I thank Jon Shultis, Carolyn Schauble and David Benson for originating many discussions of this research during its development. After visiting with Steve Brookes, Jon suggested the possibility of a connection with failure semantics. I also thank Robin Milner for providing an opportunity to visit the University of Edinburgh and discuss the research with him and his colleagues. Matthew Hennessy provided valuable feedback, including the connections to some other research, and suggestions on how the results could be presented. An anonymous referee provided several corrections and suggestions, as well as insightful comments that I have included in the abstract and introduction.

References

(1) S. Abramsky. Observation equivalence as a testing equivalence, Draft (November 1985).

(2) J.W. de Bakker, J.A. Bergstra, J.W. Klop and J.-J.Ch. Meyer. Linear time and branching time semantics for recursion with merge, in: *Automata, Languages and Programming, 10th Colloquium*, LNCS 154, (Springer-Verlag, 1983), 39-51.

(3) J.W. de Bakker, J.-J. Ch. Meyer and E.-R. Olderog. Infinite streams and finite observations in the semantics of uniform concurrency, in: *12th ICALP,* LNCS 194, (Springer-Verlag, 1985), 149-157.

(4) S.D. Brookes. On the relationship of CCS and CSP, in: *Automata, Languages and Programming, 10th Colloquium*, LNCS 154, (Springer-Verlag, 1983), 83-96.

(5) S.D. Brookes. A model for communicating sequential processes, Ph.D. Thesis, University of Oxford (1983).

(6) S.D. Brookes, C.A.R. Hoare and A.W. Roscoe. A theory of communicating sequential processes, *JACM* 31 (1984), 560-599.

(7) S.D. Brookes and A.W. Roscoe. An improved failures model for communicating processes, Carnegie-Mellon University Technical Report CMU-CS-85-112 (1985).

(8) S.D. Brookes, A.W. Roscoe and D.J. Walker. An operational semantics for CSP, prepublication copy, February 1987.

(9) M. Broy. Semantics of communicating processes, *Information and Control* 61 (1984), 202-246.

(10) E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs, *CACM* 18 (1975), 453-457.

(11) M.C.B. Hennessy. Synchronous and asynchronous experiments on processes, Internal Report CSR-125-82, Department of Computer Science, University of Edinburgh, Scotland, (September 1982).

(12) M.C.B. Hennessy. Why testing equivalence is natural, prepublication copy, August 4, 1987.

(13) C.A.R. Hoare. Communicating sequential processes, *CACM* 21 (1978), 666-677.

(14) R. Keller. Formal verification of parallel programs, *CACM* 19 (1976), 561-572.

(15) R. Milner. *A Calculus of Communicating Systems*, LNCS 92, (Springer-Verlag, 1980).

(16) R. Milner. Calculi for synchrony and asynchrony, *Theoretical Computer Science* 25 (1983), 267-310. (Computing Review 8409-0743).

(17) R. DeNicola. Testing equivalences and fully abstract models for communicating processes, Ph.D. Thesis, University of Edinburgh (November 1985).

(18) R. DeNicola and M.C.B. Hennessy. Testing equivalences for processes, in: *Automata, Languages and Programming, 10th Colloquium*, LNCS 154, (Springer-Verlag, 1983), 548-560; Expanded version in *Theoretical Computer Science* 34 (1984), 83-133.

(19) A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems, in: *Automata, Languages and Programming, 10th Colloquium*, LNCS 154, 15-32.

(20) W.C. Rounds and S.D. Brookes. Possible futures, acceptances, refusals and communicating processes, in: *22nd IEEE Symposium on Foundations of Computer Science*, (1981).

(21) M. Smyth. Powerdomains, *Journal of Computer and System Sciences* 16 (1978), 23-36.